Position Statement at the Workshop on Strategic Directions in Computing Research, MIT, Cambridge, MA, USA, June 14-15, 1996, ACM Sigplan Notices 32/1 (Jan 1997) pp. 99-102. Also published in: ACM Computing Surveys, Vol. 28, No. 4es, electronic section, December 1996, pp. 171-es, http://www.acm.org/pubs/contents/journals/surveys/1996-28/.

Future Directions in Program Transformation

Alberto Pettorossi
Department of Informatics, Systems, and Management, University of Roma II, Via della Ricerca Scientifica, 00133 Roma, Italy. adp@iasi.rm.cnr.it
and
Maurizio Proietti
IASI-CNR, Viale Manzoni 30, 00185 Roma, Italy. proietti@iasi.rm.cnr.it

---

The program transformation methodology can provide valuable techniques and tools for the development of programs from specifications and the reuse and customization of software products. Also various approaches to program optimization and program verification can be based on transformation techniques.

Program transformation is a technique for program development. It can be used both for the derivation of programs from formal specifications, and the derivation of new program versions from old program versions. Program transformation is closely related to the following important areas in the field of computer science: *semantics*, *complexity*, and *software engineering*. We want to briefly illustrate these relationships and explain some of the reasons why program transformation may profit from progress in those areas and it may also play an important role in the process of program construction. We will also identify in the area of program transformation some directions for further studies which are worth exploring in the near future.

Preservation of semantics. When deriving from a program $P1$ a new program $P2$ by transformation, we need to preserve semantics, that is, we want $Sem(P1) = Sem(P2)$ for some given semantic function Sem. Actually, we want the equivalence induced by Sem to be a congruence w.r.t. the operations which are used for building programs. This is an important engineering requirement, because for the maintenance and the enhancement of software products we have to allow for the substitution of software modules by new, better modules which satisfy equivalent specifications, and this substitution should not affect the semantics of the programs. For these reasons we look for *compositional* semantics, that is, semantics which can be defined as homomorphisms over the structure of the programs.

Moreover, in order to allow for language evolution we look for techniques for the *incremental* specification of language semantics [10]. These techniques may be very useful when, for instance, new constructs for objects, communications, and coordination of computations, are to be introduced.

Compositional and incrementally defined semantics allow for a more flexible and effective use of the transformational approach to the development of functional and logic programs [6; 11; 12; 13; 16], as well as constraint-based, object oriented, concurrent, and reactive programs.

Improvement of efficiency. When transforming a program $P1$ into a program $P2$ we want to improve performance w.r.t. a given cost function C and a relation $>$, that is, we want $C(P1) > C(P2)$. In general, in order to measure program performance one may desire to develop automatic tools for *program analysis* [17] so to evaluate, for instance, the worst-case, the average-case, and the amortized cost of the programs at hand. Through that analysis, while the program transformation is performed, the user may monitor the program improvements which have been achieved. One may also look for methods of combining calculi for semantic equivalence together with calculi for performance evaluation in the line of [1; 15].

Two approaches to program transformation. Traditionally, program transformation has been based either on the *schemata approach* [5; 18] whereby new programs are derived by instantiating given equivalences between program schemata, or the *rules + strategies approach* [4] whereby new programs are derived by sequences of applications of rules which replace program fragments by new, equivalent program fragments. The application of those rules (such as the unfold, fold, and definition rules) is controlled by strategies (such as loop fusion, tupling, generalization, abstraction, loop-absorption, accumulation) which are used for deriving programs with the desired properties. In the next years we envisage a wider development of the rules + strategies approach, because it is very flexible and it allows for the use of techniques (such as generalization and case analysis) already consolidated in the area of theorem proving and program synthesis [2; 3; 9].

Completeness of transformation rules and strategies. Various sets of semantics preserving transformation rules have been introduced for dealing with the many constructs of the different programming languages under consideration, and also various levels of control strategies have been proposed for increasing the power of the transformation systems. In the future one may want to study the particular features of such sets of rules and strategies, compare their relative strength, and possibly suggest further improvements.

In particular, one may want to design complete sets of transformation rules, in the sense that all equivalences between language expressions can be derived by applications of those rules. When looking for these completeness results, one faces some intrinsic limitations, because, for instance, even the equalities between regular expressions cannot be equationally axiomatized [14].

Also for the strategies one is interested in finding completeness results w.r.t. a given class of programs, in the sense that if there is a way of deriving from an initial program of the given class a final program with suitable properties, that final program should also be derivable by applying the transformation rules according to a strategy which is complete for that class. However, there are intrinsic limitations to the possible enhancements one may get in the general case via program transformation, and these are due to the existence of speedup theorems [7, Chap. 12].

Parametrized program transformation tools. Since the program transformation methodology can be used for the derivation of programs in different programming languages, we feel the need for the development of parametrized transformation tools which can be instantiated to particular languages, particular sets of transformation rules and strategies, and particular syntactic forms (or complexity measures) of the programs one wishes to derive. This objective requires a formal representation of the program transformation process for ensuring its automatic manipulation and possible restructuring, as it is usually done in formal systems for proof manipulation. Indeed, we can also transform programs by transforming proofs [2].

Program abstraction and program customization. Another important issue which emerges during the process of program transformation is the introduction of new basic operations and the definition of new algebras for expressing the data manipulations performed by the programs at hand. Indeed, the transformation rules may preserve the algebra in which these manipulations are expressed in the initial program, but they may also enrich or change this algebra. For instance, in the case of manipulations of natural numbers, one may transform a program which uses the zero, successor, and addition operations, into one which also uses the multiplication operation. These changes of algebra give us the ability of providing, according to the user's desires, different levels of abstraction when describing the computations performed by the program to be transformed.

Like partial evaluation [8], the development of the theory of program transformation can be extremely useful for formalizing software specialization and software customization and exploiting the properties of the contexts in which programs are used. In this respect it will also be important to propose automatic techniques for transforming collections of program modules, and not simply single modules separately, as it has been done in the past.

REFERENCES

[1] D. Archieri and E. Moggi. A denotational model for complexity. In *2nd Italian Workshop on Functional Programming*. Department of Informatics, Pisa, Italy, 1996.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Toplas*, 7(1):113–136, 1985.

[3] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[5] D. C. Cooper. The equivalence of certain computations. *Computer Journal*, 9:45–52, 1966.

[6] M. S. Feather. A survey and classification of some program transformation techniques. In L. G. L. T. Meertens, editor, *Proceedings IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, Germany*, pages 165–195. North-Holland, 1987.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[9] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. In F. L. Bauer, editor, *Logic, Algebra, and Computation*, pages 41–107. Springer-Verlag, 1991.

[10] E. Moggi. Notes on metalanguages and applications. Technical report, DISI, Genova University, Genova (Italy), 1994.

[11] R. Paige. Symbolic finite differencing - Part I. In N. D. Jones, editor, *Third European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, pages 36–56. Springer-Verlag, 1990.

[12] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[13] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[14] V. N. Redko. On defining relations for the algebra of regular events. *Ukrain. Mat. Zh.*, 16:120–126, 1964. (in Russian).

[15] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas*, 18(2):175–234, 1996.

[16] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[17] J. S. Vitter and Ph. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leuveen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 431–524. Elsevier, 1990.

[18] S. A. Walker and H. R. Strong. Characterization of flowchartable recursions. In *Proceedings 4th Annual ACM Symposium on Theory of Computing, Denver, Co., USA*, 1972.